Outline

- Host-side task and memory management
  - Recap: the GPU memory hierarchy
  - Asynchronism and streams
  - Compute capabilities
- Reusing data
  - Matrix multiplication example
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
External memory: discrete GPU

Classical CPU-GPU model

- Split memory spaces
- Highest bandwidth from GPU memory
- Transfers to main memory are slower

Ex: Intel Core i7 4770, Nvidia GeForce GTX 780
Most GPUs today

- Same memory
- May support memory coherence
  - GPU can read directly from CPU caches
- More contention on external memory

Diagram:
- CPU
- GPU
- Cache
- Main memory (8 GB)
- 26GB/s band width
Internal memory: GPU

- **Cache hierarchy**
  - Keep frequently-accessed data
  - Reduce throughput demand on main memory
  - Managed by hardware (L1, L2) or software (shared memory)
Page-locked memory

- By default, allocated memory is *pageable*
  - Can be swapped out to disk, moved by the OS...
- DMA transfers are only safe on *page-locked* memory
  - Fixed virtual → physical mapping
  - cudaMemcpy needs an intermediate copy: slower
- cudaMemcpyHost allocates page-locked memory
- Warning: page-locked memory is a limited resource!
Outline

- Host-side task and memory management
  - Recap: the GPU memory hierarchy
  - Asynchronism and streams
  - Compute capabilities
- Reusing data
  - Matrix multiplication example
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
Asynchronous execution

- By default, most CUDA function calls are *asynchronous*
  - Returns immediately to CPU code
  - GPU commands are queued and executed in-order
- Some commands are synchronous by default
  - cudaMemcpy(..., cudaMemcpyDeviceToHost)
    - Asynchronous version: cudaMemcpyAsync
- Keep it in mind when checking for errors and measuring timing!
  - Error returned by a command may be caused by an earlier command
  - Time taken by kernel<<<>>> launch is meaningless
- To force synchronization: cuThreadSynchronize()
Asynchronous transfers

- Overlap CPU work with GPU work

Can we do better?
Streams: pipelining commands

- **Command queues** in OpenCL
  - Commands from the same stream run in-order
  - Commands from different streams run out-of-order

```
CPU
  cudaMemcpyAsync HtoD, a
  cudaMemcpyAsync cudaMemcpyAsync HtoD, b
  cudaMemcpyAsync cudaMemcpyAsync HtoD, c

DMA
  copy a HtoD
  copy b HtoD
  copy c HtoD

GPU
  kernel a
  kernel b
  kernel c
```

```
cudaStreamSynchronize a
cudaStreamSynchronize b
cudaStreamSynchronize c
```
Streams: benefits

- Overlap CPU-GPU communication and computation: Direct Memory Access (DMA) copy engine runs CPU-GPU memory transfers in background
  - Requires page-locked memory
  - Some Tesla GPUs have 2 DMA engines: simultaneous send and receive

- Concurrent kernel execution
  - Start next kernel before previous kernel finishes
  - Reduces loss due to load imbalance

Example

Kernel<<<5,,,a>>>
Kernel<<<4,,,b>>>

<table>
<thead>
<tr>
<th>Serial kernel execution</th>
<th>Concurrent kernel execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>a block 0</td>
<td>a block 0</td>
</tr>
<tr>
<td>a 1</td>
<td>a 1</td>
</tr>
<tr>
<td>a 2</td>
<td>a 2</td>
</tr>
<tr>
<td>a 3</td>
<td>a 3</td>
</tr>
<tr>
<td>b 0</td>
<td>b 0</td>
</tr>
<tr>
<td>b 1</td>
<td>b 1</td>
</tr>
<tr>
<td>b 2</td>
<td>b 2</td>
</tr>
<tr>
<td>b 3</td>
<td>b 3</td>
</tr>
</tbody>
</table>
Streams: example

- Send data, execute, receive data

cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
  cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i) {
  cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
  MyKernel <<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i * size, size);
  cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
for (int i = 0; i < 2; ++i)
  cudaStreamDestroy(stream[i]);
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);

Which one is better?
Events

- Schedule synchronization of one stream with another
  - Specifies dependencies between tasks

```c
cudaEvent_t e;
cudaEventCreate(&e);
kernel1<<<,,,a>>>();
cudaEventRecord(e, a);
cudaStreamWaitEvent(b, e);
kernel2<<<,,,b>>>();

cudaEventDestroy(e);
```

- Measure timing

```c
cudaEventRecord(start, 0);
kernal<<<>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```
NVIDIA Compute capabilities

- Newer GPUs introduce additional features

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>G92</th>
<th>GT200</th>
<th>GT21x</th>
<th>GF100</th>
<th>GF104</th>
<th>GK104</th>
<th>GK110</th>
<th>GM107</th>
<th>GM204</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
<td>1.2</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
<td>5.0</td>
<td>5.2</td>
</tr>
<tr>
<td>2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Compute capability means both
  - Set of features supported
    Who can do more can do less: \( x > y \) \( \rightarrow \) CC \( x \) includes CC \( y \)
  - Native instruction set
    Not always backward-compatible
    e.g. GPU of CC 2.0 cannot run binary for CC 1.3
Compiler targets

- Compiler flags: --generate-code arch=<arch>, code=<code>, ...
  - arch=CC: directs PTX generation
    my code requires features of CC
  - code=CC: directs native code gen.
    generate code for GPU CC
  - Multiple targets possible
- CC can be
  - compute_xx for PTX
  - sm_xx for native
- Example

```
nvcc --generate-code arch=compute_10,code=sm_10
   --generate-code arch=compute_11,code='sm_12,sm_13'
   -o hello hello.cu
```
Outline

- Host-side task and memory management
  - Recap: the GPU memory hierarchy
  - Asynchronism and streams
  - Compute capabilities
- Reusing data
  - Matrix multiplication example
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
Arithmetic intensity

- Ratio of computation to memory pressure
  - High arithmetic intensity: compute bound algorithm
  - Low arithmetic intensity: memory bound algorithm

- Second lab work shows
  - NVIDIA GTX 470 needs 65 flops / word to be compute-bound

- How to reach enough arithmetic intensity?
  - Express locality: reuse values loaded from memory
Classic example: matrix multiplication

- Naive algorithm

```plaintext
for i = 0 to n-1
  for j = 0 to n-1
    for k = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
```

- Arithmetic intensity?
Classic example: matrix multiplication

- Naive algorithm

```plaintext
for i = 0 to n-1
  for j = 0 to n-1
    for k = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
```

- Arithmetic intensity: 1:1 flops/word :(  

![Diagram of matrix multiplication](image)
Reusing inputs

- Move loop on $k$ up

```plaintext
for k = 0 to n-1
    for i = 0 to n-1
        for j = 0 to n-1
            C[i,j] += A[i,k] * B[k,j]
```

- Enable data reuse on inputs
- But need to keep all matrix C on-chip!
With tiling

- Block loops on i and j

```plaintext
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    for k = 0 to n-1
      for i2 = i to i+15
        for j2 = j to j+15
          C[i2,j2] += A[i2,k] * B[k,j2]
```

- For one block: product between horizontal panel of A and vertical panel of B

![Diagram of A, B, and C matrices with tiling]

Constant size
With more tiling

- Block loop on k

```plaintext
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    for k = 0 to n-1 step 16
      for k2 = k to k+15
        for i2 = i to i+15
          for j2 = j to j+15
            C[i2,j2] += A[i2,k] * B[k,j2]
```

![Diagram](image)
for $i = 0$ to $n-1$ step 16
  for $j = 0$ to $n-1$ step 16
    $c = \{0\}$
    for $k = 0$ to $n-1$ step 16
      $a = A[i..i+15,k..k+15]$  
      $b = B[k..k+15,j..j+15]$  
      for $k2 = 0$ to 15
        for $i2 = 0$ to 15
          for $j2 = 0$ to 15
            $c[i2,j2] += a[i2,k2] \times b[k2,j2]$  
      $C[i..i+15,j..j+15] = c$
Breaking into two levels

Run loops on i, j, i2, j2 in parallel

```plaintext
for // i = 0 to n-1 step 16
    for // j = 0 to n-1 step 16
        c = {0}
        for k = 0 to n-1 step 16
            a = A[i..i+15,k..k+15]
            b = B[k..k+15,j..j+15]
            for k2 = 0 to 15
                for // i2 = 0 to 15
                    for // j2 = 0 to 15
                        c[i2,j2] += a[i2,k2] * b[k2,j2]
            C[i..i+15,j..j+15] = c
```

Let's focus on threads
Level 1: SIMD (PRAM-style) version

- Each processor has ID (x,y)
  - Loops on i2, j2 are implicit

\[
c[x,y] = 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
a[x,y] = A[i+x,k+y] \\
b[x,y] = B[k+x,j+y]
\]

\[
\text{for } k2 = 0 \text{ to } 15 \\
c[x,y] += a[x,k2]*b[k2,y]
\]

\[
C[i+x,j+y] = c[x,y]
\]

Private writes: no conflict

Load submatrices a and b
Multiply submatrices \( c = a \times b \)
Store submatrix c

Read from other processors

- How to translate to SPMD (BSP-style) ?
SPMD version

- Place synchronization barriers

\[
\begin{align*}
  c[x,y] &= 0 \\
  &\text{for } k = 0 \text{ to } n-1 \text{ step 16} \\
  a[x,y] &= A[i+x,k+y] \\
  b[x,y] &= B[k+x,j+y] \\
  \text{Barrier} \\
  &\text{for } k2 = 0 \text{ to } 15 \\
  c[x,y] &= a[x,k2]*b[k2,y] \\
  \text{Barrier} \\
  C[i+x,j+y] &= c[x,y]
\end{align*}
\]

- Why do we need the second barrier?
Data allocation

- 3 memory spaces: Global, Shared, Local

- Where should we put: A, B, C, a, b, c?

```
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y] += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```
Data allocation

- Memory spaces: **Global**, **Shared**, **Local**
  - As local as possible

```plaintext
    \[ c = 0 \]

    for k = 0 to n-1 step 16
      \[ a[x,y] = A[i+x,k+y] \]
      \[ b[x,y] = B[k+x,j+y] \]
      Barrier
      for k2 = 0 to 15
        \[ c += a[x,k2]*b[k2,y] \]
      Barrier
      \[ C[i+x,j+y] = c \]
```

- **Local**: private to each thread (indices are implicit)
- **Global**: shared between blocks / inputs and outputs
- **Shared**: shared between threads, private to block
CUDA version

- Straightforward translation

```c
float Csub = 0;

for(int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    __syncthreads();
    for(int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

matrixMul example: cuda/samples/0_Simple/matrixMul

Precomputed base addresses
Declare shared memory
Linearized arrays
Outline

- Host-side task and memory management
  - Recap: the GPU memory hierarchy
  - Asynchronism and streams
  - Compute capabilities
- Reusing data
  - Matrix multiplication example
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
Parallel regularity

- Similarity in behavior between threads

<table>
<thead>
<tr>
<th>Control regularity</th>
<th>Thread</th>
<th>Irregularity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

```
int i=21;
switch(i) {
    case 2:...
    case 17:...
    case 21:...
}
```

<table>
<thead>
<tr>
<th>Memory regularity</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Data regularity</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=32</td>
<td>a=32</td>
<td>a=17</td>
</tr>
<tr>
<td>b=52</td>
<td>b=52</td>
<td>b=15</td>
</tr>
<tr>
<td>a=32</td>
<td>a=32</td>
<td>a=-5</td>
</tr>
<tr>
<td>b=52</td>
<td>b=52</td>
<td>b=0</td>
</tr>
<tr>
<td>a=32</td>
<td>a=32</td>
<td>a=11</td>
</tr>
<tr>
<td>b=52</td>
<td>b=52</td>
<td>b=-2</td>
</tr>
<tr>
<td>a=32</td>
<td>a=32</td>
<td>a=42</td>
</tr>
<tr>
<td>b=52</td>
<td>b=52</td>
<td>b=52</td>
</tr>
</tbody>
</table>
Memory access patterns

In traditional vector processing

On GPUs

- Every load is a gather, every store is a scatter
Breakdown of memory access patterns

- Vast majority: uniform or unit-strided
  - And even aligned vectors

“In making a design trade-off, favor the frequent case over the infrequent case.”
Memory coalescing

- In hardware: compare the address of each vector element
- Coalesce memory accesses that fall within the same segment

Unit-strided requests → One transaction

Irregular requests → Multiple transactions

- Dynamically detects parallel memory regularity
Coalescing concurrent requests

- **Unit-strided detection (NVIDIA CC 1.0-1.1 coalescing)**
  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

- **Minimal coverage (NVIDIA CC 1.2 coalescing)**
  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

Impact on work distribution, data structures?
Consequences: threading granularity

- Coarse-grained threading
  - **Decouple** tasks to reduce **conflicts** and inter-thread communication
  - e.g. MPI, OpenMP

- Fine-grained threading
  - **Interleave** tasks
  - Exhibit **locality**: neighbor threads share memory
  - Exhibit **regularity**: neighbor threads have a similar behavior
  - e.g. CUDA, OpenCL
Array of structures (AoS)

- Programmer-friendly memory layout
  - Group data logically
- Memory accesses not coalesced
  - Bad performance on GPU

need to rethink data structures for fine-grained threading
Structure of Arrays (SoA)

- Transpose the data structure
  - Group together similar data for different threads
- Benefits from memory coalescing
  - Best performance on GPU

```c
struct Image {
  float R[480][640];
  float G[480][640];
  float B[480][640];
};
Image image_SoA;

kernel void luminance(Image img,
  float luma[][]) {
  int x=tid.x; int y=tid.y;
  luma[y][x]=.59*img.R[y][x]
    + .11*img.G[y][x]
    + .30*img.B[y][x];
}
```
Outline

- Host-side task and memory management
  - Recap: the GPU memory hierarchy
  - Asynchronism and streams
  - Compute capabilities
- Reusing data
  - Matrix multiplication example
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
Vector loads

- We can load more data at once with vector types
  - float2, float4, int2, int4...
  - More memory parallelism
  - Allows to reach peak throughput with fewer threads

Multiple outstanding loads

- Multiple independent loads from the same thread can be pipelined
  - More memory parallelism
  - Peak throughput with yet fewer threads

```c
__global__ void luminance(Image img, float luma[][[]]) {
    int x=threadIdx.x, y=threadIdx.y;
    luma[y][x]=.59*img.R[y][x]
    + .11*img.G[y][x]
    + .30*img.B[y][x];
}
```
Buffer accesses through shared memory

- Global memory accesses are the most expensive
  - Focus on optimizing global memory accesses
- Strategy: use shared memory as a temporary buffer

1. Load with regular accesses
2. Read and write shared memory with original pattern
3. Store back to global memory with regular accesses
Example: matrix transpose

- $B = A^T$

- Naive algorithms
  - Option 1
    - Thread $i, j$:
      - $B[j, i] = A[i, j]$
  
  - Option 2
    - Thread $i, j$:
      - $B[i, j] = A[j, i]$

- Which one is better?
  - What is the problem?
Example: matrix transpose

- \( B = A^T \)
- Naive algorithms
  - Option 1
    - Thread \( i,j: \)
      \[ B[j,i] = A[i,j] \]
    - Coalesced
  - Option 2
    - Thread \( i,j: \)
      \[ B[i,j] = A[j,i] \]
    - Non-coalesced

- Both are equally bad
  - Access to one array is non-coalesced
Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back

Example with 16×16 blocks

Block bx,by, Thread tx,ty:
\[
a[ty,tx] = A[by*16+ty,by*16+tx]
\]
Syncthreads
\[
b[ty,tx] = a[tx,ty]
\]
Syncthreads
\[
B[by*16+ty,bx*16+tx] = b[ty,tx]
\]
Objection

- Isn't it just moving the problem to shared memory?
- Yes: shared memory has access restrictions too
- But
  - Shared memory is much faster, even for irregular accesses
  - We can optimize shared memory access patterns too
Outline

- Host-side task and memory management
  - Recap: the GPU memory hierarchy
  - Asynchronism and streams
  - Compute capabilities
- Reusing data
  - Matrix multiplication example
- Work partitioning and memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
Inside each SM, shared memory is distributed between multiple banks
- 16 or 32 banks
Shared memory bank assignment

- Interleaved on a word-by-word basis: Modulo placement of data

Shared memory address space

0

Bank 0

Bank 1

Bank 2

Bank 3

Actually 16 (or 32) banks

0xffffffff
Shared memory: the good

- Threads access contiguous locations: no conflict
  - All threads can be served concurrently
Shared memory: the bad

- Threads access random locations: some conflicts
  - Some threads have to wait for a bank
Shared memory: the ugly

- Threads access locations spaced by 16: systematic conflict
  - All threads have to wait for the same bank
Example: matrix transpose

- Where are bank conflicts?

Block \( bx, by, \) Thread \( tx, ty: \)
- \( a[ty\times16+tx] = A[by\times16+ty, by\times16+tx] \)
  - Syncthreads
- \( b[ty\times16+tx] = a[tx\times16+ty] \)
  - Syncthreads
- \( B[by\times16+ty, bx\times16+tx] = b[ty\times16+tx] \)
Example: matrix transpose

- Where are bank conflicts?

  Block \( bx, by, \) Thread \( tx, ty: \)
  
  \[
  a[ty*16+tx] = A[by*16+ty, by*16+tx]
  \]
  
  Syncthreads

  \[
  b[ty*16+tx] = a[tx*16+ty]
  \]
  
  Syncthreads

  \[
  B[by*16+ty, bx*16+tx] = b[ty*16+tx]
  \]

- How to avoid them?
Remapping data

- Solution 1: pad with empty cells

Block bx, by, Thread tx, ty:
\[ a[ty*17+tx]=A[by*16+ty, by*16+tx] \]
Sync threads
\[ b[ty*16+tx]=a[tx*17+ty] \]
Sync threads
\[ B[by*16+ty, bx*16+tx]=b[ty*17+tx] \]

- No bank conflicts
- Memory overhead
Remapping data

- Solution 2: different mapping function
  - Example: map \([y,x]\) to \(y\times16 + (x+y \mod 16)\)
  - Or \(y\times16 + (x \land y)\)

Block \(bx,by\), Thread \(tx,ty\):
\[
a[ty\times16+(tx+ty)\mod 16]=A[by\times16+ty,by\times16+tx]
\]
Sync threads
\[
b[ty\times16+tx]=a[tx\times16+(ty+tx)\mod 16]
\]
Sync threads
\[
B[by\times16+ty,bx\times16+tx]=b[ty\times17+tx]
\]

- No bank conflicts
- No memory overhead
Recap

- Overlap long-latency communications with computations
- Avoid global accesses when you can
  - Reuse data to get enough arithmetic intensity
  - Use registers and shared memory whenever possible
- Make consecutive threads access contiguous data
  - Stage data in shared memory if needed
- Avoid bank conflicts in shared memory
- Express locality and regularity
Conclusion

- We have focused on the most important factor: memory accesses
- Next lecture
  - Optimizing matrix multiplication accesses
  - Workload partitioning: how many blocks, threads?
  - Instruction-level optimizations
- Lab work:
  continuation of ln(2) series computation